Project 9: Intermediate Code Generation (Part 2)

Due Date: Tuesday, February 14, 2006, Noon
Duration: One week

Overview

In this project, you will modify and add to the **Generator.java** file.

You will add the ability to generate IR code for:

- All boolean-valued operators, i.e.,
  <, <=, >, >=, =, <>, AND, OR, NOT
- The following kinds of statements:
  IF, WHILE, LOOP, FOR, RETURN, ASSIGNMENT, CALL, EXIT

Code generation for the following will be done in a later project:

- READ statement
- WRITE statement
- Array accessing
- Array constructors
- Record accessing
- Record constructors

Part of the challenge of this project is that you must generate "short-circuit" code for the boolean operators.

Files

Start by creating a new directory for this project, called **p9**.

Copy all files (except the **tst** directory) from the last project into this new directory. In other words, keep an intact copy of your **p8** files (with Unix timestamp unmodified).

Here are the new files for this project:

The assignment PDF file (this document)

tst
Contains several new test programs, plus all test programs from the previous project. The expected output files, even from the old test programs, are different. The **call** test has also changed a little.

runAll
Modified to include the new tests.

IR.java
Modified slightly to print out **MaxLexicalLevel**.

Main.jar
Modified.

Short-Circuit Operators

In PCAT, some variables can have type "boolean"; how are the values TRUE and FALSE stored? In our compiler, we will store Booleans using a full word (i.e., 32-bits). TRUE will be represented as 0x00000001 and FALSE will be represented as 0x00000000.

What is short-circuit evaluation? It means that the second operand of a logical operator will not be evaluated if the resulting value can be determined after the first operand has been evaluated.

Consider the logical AND operator. If the first operand turns out to be FALSE, then there is no need to evaluate the second operand, since the result of the AND will be FALSE in any case. Furthermore, the language specification says that the second operand *must not* be evaluated in this case.

If the first operand of the AND expression is TRUE, then the second operand must be evaluated. Notice that, once we have found that the first operand is TRUE, the value of the entire AND expression will be whatever the value of the second operand is.

Next consider the logical OR operator. If the first operand turns out to be TRUE, then there is no need to evaluate the second operand, since the result of the OR will be TRUE in any case. In this case, the second operand *must not* be evaluated. However if the first operand is FALSE, then the value of the entire expression will be whatever the second operand turns out to be.

[Note that for logical XOR and logical EQUALS, short-circuit operation is impossible. Regardless of what the first operand turns out to be, the second operand must be evaluated. Are there other logical operators for which short-circuit evaluation can be implemented?]

As many programmers know, short-circuit operation can be useful and programmers will often depend on it being there in the code they write. Consider this C code:

```
if (i >= 0 && a[i] != 999) ... ;
if (p && p.next) ... ;
```

You must modify **genExpr** (and some of the methods it calls) to generate code for all remaining operators, handling short-circuit evaluation correctly.

To deal with relational operators and with Boolean expressions, the following IR instructions may be used:

```
IR.label      label:
IR.go_to      goto label

IR.gotoiEQ    if arg1=arg2  then goto label   (integer)
IR.gotoiNE    if arg1!=arg2 then goto label   (integer)
IR.gotoiLT    if arg1<arg2  then goto label   (integer)
IR.gotoiLE    if arg1<=arg2 then goto label   (integer)
IR.gotoiGT    if arg1>arg2  then goto label   (integer)
IR.gotoiGE    if arg1>=arg2 then goto label   (integer)

IR.gotofEQ    if arg1=arg2  then goto label   (float)
IR.gotofNE    if arg1!=arg2 then goto label   (float)
IR.gotofLT    if arg1<arg2  then goto label   (float)
IR.gotofLE    if arg1<=arg2 then goto label   (float)
IR.gotofGT    if arg1>arg2  then goto label   (float)
IR.gotofGE    if arg1>=arg2 then goto label   (float)
```

Note that we use integer comparison instructions for integer, pointer, and Boolean values and we use float comparison instructions for comparing real values.

Constants 0, 1, and 4

There are a couple of places where you will need the integer constants 0 and 1. At the beginning of the **generateIR()** you might want to create two **Ast** nodes of type **IntegerConst** and fill them in with **iValue**s equal to 0 and to 1. (In my own code, I called them **constant0** and **constant1**.) These can then be used in handling **BooleanConst**s and **constant0** can be also used for **NilConst**s as well.

By creating these constant nodes once at the beginning, right before you call **genBody()** for the main body, you won't have to mess with creating them later and you will also have only one copy of each of these nodes, which can be shared by different parts of the AST.

In the next project, when we are generating the code for array dereferencing address calculations, we will also need an **IntegerConst** with **iValue** equal to 4. You might as well call it **constant4** and create it when you create **constant0** and **constant1**.

Hints for Implementing Short-Circuit Code Generation

The method **genExpr()** will be called to generate code for Boolean expressions. In some cases, **genExpr()** must generate code to compute a value (either 0 or 1), move it into a temporary, and return that temporary as the "place" where the value is.

For expressions such as in the following example, **genExpr()** will need to generate code to move the value into some other location (e.g., "x").

```
x := a<b and (c<d or e<f);
```

In other cases, **genExpr()** must generate code to jump somewhere. The actual Boolean value is not needed and should not be produced. In the following example, **genExpr()** should generate code to jump either to the first "then" statement or to the first "else" statement.

```
if a<b and (c<d or e<f) then ... else ... end;
```

In order to get **genExpr()** to generate the correct sort of code, you must pass it an indication of whether or not you need code ending with branches and, if so, where the code should branch to. You can do this by adding 2 additional parameters to **genExpr()**, called **trueLabel** and **falseLabel**.

The **trueLabel** and **falseLabel** arguments will either

(1) both be NULL, or
(2) point to Java Strings, such as "Label_23" and "Label_51", giving branch targets.

(We should never have the situation where one of **trueLabel** / **falseLabel** is NULL and the other is non-NULL.)

I recommend modifying the prototype for **genExpr()** to:

```
Ast.Node genExpr (Ast.Node t, String trueLabel, String falseLabel)
        throws FatalError
```

Sometimes the arguments **trueLabel** and **falseLabel** will be NULL. In that case, **genExpr()** must generate code to evaluate the expression into a variable and return that.

Other times, **trueLabel** and **falseLabel** will be labels (i.e., Java Strings). In that case, **genExpr()** must generate code to jump to **trueLabel** when the expression evaluates to TRUE, and to **falseLabel** otherwise. (When **trueLabel** and **falseLabel** are not NULL, the value returned from **genExpr()** will be ignored. Your **genExpr()** can just return NULL.)

When **genExpr()** is generating code for a non-Boolean expression, **trueLabel** and **falseLabel** will always be NULL. So if **trueLabel** and **falseLabel** are non-NULL, **genExpr()** must be generating code for a Boolean expression. But for some Boolean expressions, **trueLabel** and **falseLabel** will still be NULL.

You'll also need to modify the following methods to add the **trueLabel** and **falseLabel** parameters:

genBinaryOp()
genUnaryOp()
genFunctionCall()

If **genExpr()** is passed **trueLabel=falseLabel**=NULL, then it must generate whatever code is necessary to compute the Boolean value (either 0 or 1) and get it into some variable. **GenExpr()** will then return that variable as the "place" where the value is. This variable could be a temporary variable created by **genExpr()** or it could be a "regular" variable of type Boolean.

GenExpr() may have to generate a bunch of instructions to compute the value and move it into a new temporary variable but, if the expression happens to be a simple Boolean variable, then **genExpr()** can just return this variable itself without generating any instructions.

Regardless of how complicated the expression is and how much code is generated, if **genExpr()** is passed **trueLabel=falseLabel=**NULL then it will return a pointer to the **VarDecl** or **Formal** node that describes the variable containing the result.

On the other hand, if **genExpr()** is passed branch target labels (that is, if **trueLabel** and **falseLabel** are not NULL), then **genExpr()** must generate code that will end by jumping. **GenExpr()** should not return anything. (It should just return NULL.) Whichever method invoked **genExpr()** will be expecting the "answer" in the form of branches and will therefore ignore any returned value.

How should **genExpr()** generate code for various kinds of expressions, like **IntegerConst**, **FunctionCall**, and **BinaryOp**?

Consider generating code for **IntegerConst**. (Since this case will be straightforward, it is probably better to handle it directly within **genExpr**, instead of creating a "**genIntegerConst**" method.) If the expression is an integer constant, **genExpr** will never be passed **trueLabel** and **falseLabel** since it is not a Boolean, so you don't even need to test them against NULL. For the **IntegerConst** node, you should create a temp, generate code to move the constant into that temp, and then return the temp (or more specifically, return a ptr to the **VarDecl** for that temp).

Next consider **genFunctionCall()**. First, it will generate the code to call the function. But then it must check to see if **trueLabel** and **falseLabel** are not NULL. (After all, the called function might return a Boolean and you might need to generate branches.) If you have **trueLabel** and **falseLabel**, then **genFunctionCall()** must generate instructions to test the returned value and branch accordingly. Otherwise it will put the return value in a temporary and return that.

Next consider **genBinaryOp()**. You may be tempted to immediately call **genExpr()** twice to deal with the sub-expressions before switching on the **op** field. Don't! You should switch on the **op** field first. Then, you can recursively invoke **genExpr** in each case, as appropriate for that case.

For the arithmetic operators (like '+'), you can call **genExpr()** twice on the sub-expressions, passing NULL **trueLabel** and **falseLabel** down.

For an operator like OR or AND, you must first test to see if you have a **trueLabel** and **falseLabel**.

If you do have branch labels, then generating the code is fairly short. You'll have to call **genExpr()** twice, once for each of the arguments. Furthermore, you'll need to pass the branch labels down into the recursive invocations in a slightly different order to achieve the short-circuit behavior.

On the other hand, if you don't have labels for AND or OR, it means **genExpr** is required to place a 0 or 1 into a variable and return it. You'll still need to call **genExpr** recursively to deal with the two operand sub-expressions but, since the sub-expressions could involve other Boolean operators and since short-circuit evaluation is required for the sub-expressions, things get a little tricky.

Let's imagine that **genExpr()** is called on an OR expression and that **trueLabel** and **falseLabel** are NULL. I recommend that you first create a temporary and two new labels. Next, call **genExpr()** once on the same expression (i.e., on the very same OR expression that you are in the middle of trying to generate code for). [Fortunately, **genExpr()** will generate the right code, since the labels will be present in the recursive call, the OR will be handled by the code described two paragraphs earlier!] After this recursive call to **genExpr()**, you can then generate several IR instructions to move 0 and to move 1 into the temp, using the labels and gotos. Finally **genExpr()** will return the temp. Neat, huh?

In the case of the relational operators (like <), begin by calling **genExpr()** twice to generate code for the sub-expressions. (For the sub-expressions, you should not pass any branch labels down. After all, the sub-expressions are integers or reals, so short-circuit evaluation for them is not even possible.) Then, test to see whether you have branch targets (i.e., test if **trueLabel** and **falseLabel** are non-null) and generate code differently in each case.

Main Entry and Exit

The first IR instruction generated for the "main" body should be a **mainEntry** instruction. Following the last instruction of the main body, the **mainExit** instruction should be generated. From these, we will ultimately generate something like:

mainEntry...

```
                .global  main
    main:       save     %sp,-???,%sp      ! set up main frame
```

mainExit...

```
                mov      1,%g1             ! exit request
                ta       0                 ! trap to system
```

When we generate the **save** instruction, we will need to know the **frameSize** of the main body. (In the next project, we will compute this number and store it in the **Body** node.) The **mainEntry** instruction has a parameter which points to the **Body** node so that we can follow it later when we generate the SPARC instructions.

When **printIR** prints the **mainEntry** instruction, it will also go to the **Body** node and print the value of **frameSize**. You will not set the **frameSize** until a later project, so it will print as just zero.

[The **Main.jar** black box program will also print the **frameSize** as zero, even though the black box program contains the code generation methods for later proijects; I've commented out the **frameSize** computations in the version of the black box you'll be using for this project.]

The **mainExit** instruction has no parameters.

Code for Procedures

To generate the code for a procedure, you will need to generate a **procEntry** instruction, followed by several **formal** instructions. For example, for the source code:

```
    procedure foo (x, y, z: real) : real is
      begin
        ...
        return b;
      end;
```

you must generate the following IR instructions:

```
! PROCEDURE...
            procEntry foo,lexLevel=1,frameSize=0
            formal 1,x
            formal 2,y
            formal 3,z
...
! RETURN...
            returnExpr b
```

The **procEntry** IR instruction has one operand which you should set to point to the **ProcDecl** node of the procedure. When this instruction is printed, **printIR** will go to the **ProcDecl** node and print the value of **lexLevel**. **PrintIR** will also go to the **Body** node and print the value of **frameSize**. [**LexLevel** was set by the **Checker**. **FrameSize** will be zero for this project and will be set in a later project.]

Whenever you encounter a RETURN statement, you should generate a "return" instruction. Actually, there are two IR return instructions, **returnExpr** and **returnVoid**, depending on whether we are returning a value from a function or simply returning from a procedure. In the case of **returnExpr**, there is a single operand which should be set to point to a variable containing the value to be returned. This variable (which is usually a temporary) will have been returned from a call to **genExpr()**.

In the source file, a procedure may occur between the variable initializations and the statements in a body. Obviously, we need to generate the IR code for a procedure somewhere else besides right in the middle of the IR instructions for its enclosing procedure. Assume that procedures **foo1**, **foo2**, and **foo3** were defined (i.e., directly nested) within procedure **bar**. The IR code for procedures **foo1**, **foo2**, and **foo3** should therefore be generated directly after the IR code for the statements for the body of **bar**.

Loops and Exits

In PCAT, a WHILE, FOR, or LOOP statement may include an EXIT statement anywhere in its body. When you generate the code for the EXIT statement, you'll need to generate a **goto** instruction to the point just after the looping statement. You'll need to generate a label—the "exit label"—directly after each looping statement to serve as the target of any EXIT branches that appear in the looping statement. This is the label that any EXIT statements in the loop will branch to.

To accommodate this, a field called **exitLabel** has been added to the **WhileStmt**, **ForStmt**, and **LoopStmt** nodes. When you generate code for a **WhileStmt**, **ForStmt**, or **LoopStmt**, you can simply store the exit label here.

Fortunately we have already linked each **ExitStmt** node to the loop it is exiting from. The **myLoop** field in the **ExitStmt** has already been set (during type-checking) to point to either a **WhileStmt**, **ForStmt**, or **LoopStmt**. So to generate code for an **ExitStmt**, you can easily figure out the label to put into the **goto** instruction.

New Global Variables

In project 5 or 6 (when we wrote **Checker.java** and we used **openScope** and **closeScope**), we talked about the lexical level. There was a static variable (called **level**) which was incremented by **openScope** and decremented by **closeScope**.

In this project, you will need to re-compute the lexical level as you walk the AST, but this should be straightforward. I have included a field called **lexicalLev** in **Generator0.java** for this purpose.

Here are the fields I included in the starter file:

```
int lexicalLev = 0;
int maxLexicalLevel = 0;
Ast.Body currentBody;
Ast.StringConst stringList = null;
Ast.RealConst floatList = null;
int nextLabelNumber = 1;
int nextTempNumber = 1;
```

You must update the values of **lexicalLev**, **maxLexicalLevel**,

You should already be modifying **currentBody** in the last project.

The variables **stringList** and **floatList** are for the next project; don't worry about them now. The variables **nextLabelNumber** and **nextTempNumber** are used in the **newLabel** and **newTemp** methods; you should not need to do anything concerning them either.

The variable **lexicalLev** is initialized to 0. Every time you enter a new **ProcDecl**, you should increment it and, when you finish generating the code for a **ProcDecl**, you should decrement it.

Note that you can test whether **lexicalLev = 0** within any **Body** to see whether this is the "main" body or not. You should generate the **mainEntry** and **mainExit** instruction within the **genBody()** method, and not within **generateIR()**. Why?

You must set **maxLexicalLevel** to the maximum degree of procedure nesting in the program. So every time you increment **lexicalLev**, simply check to see if this is a new maximum and update **maxLexicalLevel** if so. [We'll need to know the maximum degree of procedure nesting when we use "display registers," which will be discussed later.]

Grading

The primary consideration for grading will be correctness. The output of your program will be compared to the output produced by the "black box" program, **Main.jar**. Your output should match exactly.

Your code should also be well organized and clearly documented.

Be sure to follow my style guidelines for commenting and indenting your Java code. There is a link on the class web page called "Coding Style for Java Programs." Please read this document. Also look at the Java code I am distributing for examples of the style we are using in this class.

During testing, the grader will compile your **Generator.java** file and link it with my files, including my **Lexer.class**, **Parser.class**, **Checker.class**, and **PrettyPrint.class**.

[IF YOU DIDN'T TAKE CS-321 LAST TERM, IGNORE THE NEXT PARAGRAPH...]

I encourage you to use your own files during testing, but I also strongly encourage you to test your **Generator.java** with my **Lexer.class**, **Parser.class**, **Checker.class**, and **PrettyPrint.class**, just to make sure it works correctly with them. While there should be no difference, it still seems like a good idea.

Standard Boilerplate...

It is considered cheating to decompile or look inside any **.class** or **.jar** file I provide. If you have questions about what these files do, please ask me!

As before, email your completed program as a plain-text attachment to:

```
cs321-01@cs.pdx.edu
```

Don't forget to use a subject like:

```
Proj 9 - John Doe
```

DO NOT EMAIL YOUR PROGRAM TO THE CLASS MAILING LIST!!!

Your code should behave in exactly the same way as my code. If there is any question about the exact functionality required,

(1) Use my code (the "black box" **.jar** file) on test files of your own creation, to see how it performs.

(2) *Please ask* or talk to me!!! I will be happy to clarify any of the requirements.

Do not submit multiple times.

Please keep an unmodified copy of your file on the PSU Solaris system with the timestamp intact. This is required, in case there are any "issues" that arise after the due date.

In other words: **DO NOT MODIFY YOUR "Generator.java" FILE AFTER YOU SUBMIT IT**. You can create a **p10** directory, copy all files over and keeping working, if you need to.

Work independently: you must write this program by yourself.